

SESSION: 2018/19 DIET 1

Mobile Games Development 1 Coursework: Code Explanation

GitHub Link: <https://github.com/CMcLuskie/MGD-Coursework/tree/MGD1-Submission-2018.->

|  |  |  |
| --- | --- | --- |
| Student Name | Student ID | Student E-mail |
| Conall McLuskie | S1509449 | Cmclus200@caledonian.ac.uk |

Student Declaration

I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award.

Signature: Conall McLuskie Date:

Contents

[Introduction 3](#_Toc532322347)

[Game Overview 3](#_Toc532322348)

[CSS 3](#_Toc532322349)

[HTML 3](#_Toc532322350)

[Main Activity 4](#_Toc532322351)

[Game Framework 5](#_Toc532322352)

[Main Game 5](#_Toc532322353)

[Collision Detection 5](#_Toc532322354)

[Events 6](#_Toc532322355)

[Controls 6](#_Toc532322356)

[Game States 7](#_Toc532322357)

[Canvas 7](#_Toc532322358)

[Rendering 7](#_Toc532322359)

[Sound 8](#_Toc532322360)

[Inheritance 9](#_Toc532322361)

[Sprite 9](#_Toc532322362)

[Background 9](#_Toc532322363)

[Player 10](#_Toc532322364)

[Enemy 10](#_Toc532322365)

[Coin 10](#_Toc532322366)

[Game Logic 11](#_Toc532322367)

[Initializing 11](#_Toc532322368)

[State Machine 11](#_Toc532322369)

[Play State 11](#_Toc532322370)

[Appendix 13](#_Toc532322371)

[References 18](#_Toc532322372)

# Introduction

## Game Overview

The game being developed in this project is an infinite falling game where the player will be parachuting down while avoiding birds and collecting coins. The Player controls the game by rotating the device parallel to the screen (on mobile) or using the arrow or wasd keys (on pc). This section will discuss the CSS, HTML, and Main Activity files. These files are necessary to run the game in a browser either on desktop or mobile.

## CSS

The Cascading Style Sheet (CSS) dictates how the information in the HTML file is displayed. The canvas element and body of the HTML is set to take up 1005 of the window space without any padding or margin. The overflow has been set to ensure that if anything appears outside of the body of the window then it is not rendered.

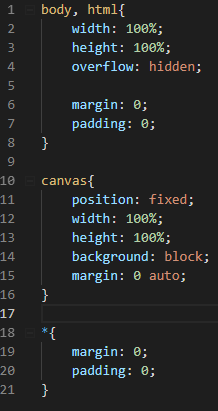


Figure CSS File

The canvas element’s position is fixed to ensure that it stays in place. If the page were to scroll for some reason, then the canvas would not move and stay in place relative to the viewport.

## HTML

Figure 2 shows the entire HTML file for the game. This file links all other elements of the game together. The title element is what is displayed in the tab title in a browser. The CSS files is linked using the link element and the Script element links the main game script to enable the game to run. The first function to be called is the load function which initializes all the code in the java script (js) files. It is also worth noting that the canvas is given the id of *gameCanvas.* This will be needed later when the js files need to reference the canvas.



Figure HTML File

## Main Activity

The Main Activity is a Java file that will get and set the low-level values for the window. Additionally, the Java file for sound is linked in this script. A notable part of this method is the UI Settings as shown in figure 3. It is here where the game is put into full screen and the navigation bar for android is hidden to maximise screen space. Another important function is the setRequestedOrientation function which is where the game’s orientation is set to portrait. This is required as it changes where the top left corner is, and therefore where the game gets rendered from.

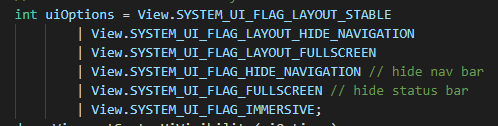


Figure - Setting UI Options

# Game Framework

## Main Game

The main game script is the script that is the link between the game’s html and all the other js files used to code the game. This is done by using the write function as shown in figure 4.

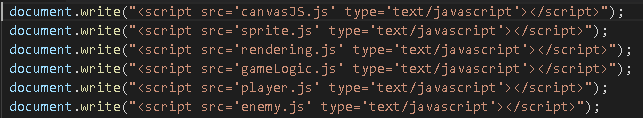


Figure - Write function

The load function that was linked in the html file initializes the initial values needed for the game and begins the game loop. The game loop acts as the functional update where delta time is calculated, and each individual update functions are called as shown in figure 5. Separation of the update functions ensures that the code is kept tidy and in the correct place. The function is then looped using the requestAnimationFrame function.

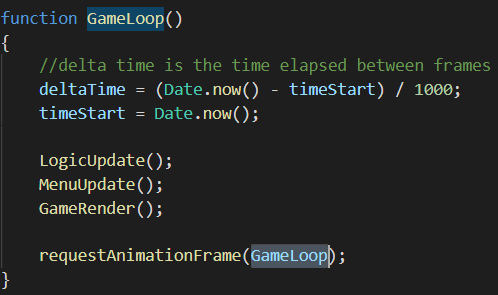


Figure - Functional Game Loop

### Collision Detection

The collision detection used in this game is a bounding box collision. Inside the sprite class a function hat returns a rectangle acts as a collider. The values needed for this is the x & y positions, and the width and height of the collider. The x and y position had to be dynamic to account for the movement of the objects. Therefore, when initialising the rectangle, the position values passed in to create the collider is an offset that gets added to the position values of the sprite. For example, if the collider was to begin at the top left of the sprite, then an offset of 0 would be passed in.

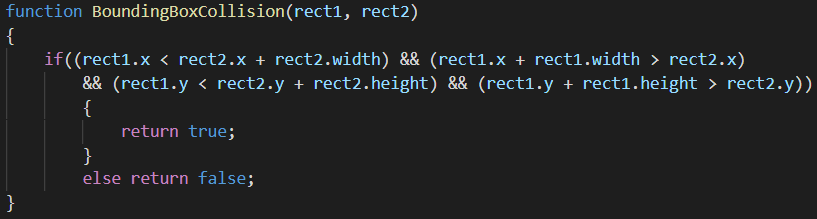


Figure - Collision Detection Function

The actual collision detection, as mentioned above, is bounding box. This means that each value of two rectangles are compared to see if they are colliding. Figure 6 shows the collision detection method, and as you can see if the first rectangle’s positions (plus width or height) are between the second rectangle’s position plus the width or height.

To test the rectangles during development they were drawn with the built in drawRect function, which takes the same values as needed for the rectangle and draws them to the canvas. This allowed visualisation of the rectangle. Values were changed in the console and noted to save time by not swapping back to the code editor constantly.

### Events

Several events are throughout the game’s code. However, to use these event listeners need to be added. figure 7 shows the function that does this. The string being passed into the addEventListener function is the event that is being listened for, where the next parameter is the custom function that holds the behaviours required for each event.



Figure - Adding Event Listeners

This is mainly used for input. However, whenever the window is resized the canvas also gets resized to fit the new window size.

### Controls

During testing, before testing android, it was tested in the browser. To avoid changing the game logic code further in development Booleans were used to determine what each method of input was doing. Keyboard input is shown in figure 8. Each keypress would set a Boolean to true, and on key up would set it to false. The *which* keycode was used to determine what key was being pressed.

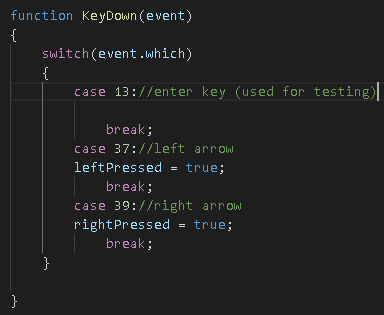


Figure - Listening for Keyboard Input

Although keyboard input does function on an android device it is not the most suitable control scheme for the platform. Therefore, the game also uses the device’s rotation to control the character.

This is done by setting a limit of how sensitive the controls should be, as figure 9 shows a value of 5 was used. The first if statement checks the dead-zone of rotation to ensure the player’s sprite isn’t moving when the player doesn’t want it to be. The devices rotation does not use x, y, and z notation; it instead uses the alpha, beta, and gamma notation. The beta value was used as it is the rotation parallel to the phone’s display. Appendix A shows this in a diagram. The result for the movement ends up with the player’s sprite falling to the bottom of the screen mimicking real world gravity.

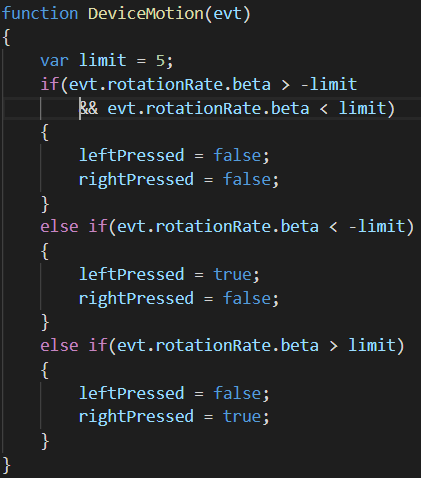


Figure - Listening for the Devices Rotation

## Menu

The menu manager script is responsible for initializing any buttons coded in the game. Therefor it initializes the button in the start menu. To detect input from the button a 1x1 collision box is created where the player clicks (or touches) and whenever the player clicks a collision is checked between the button and the click.

## Game States

Game states are needed for the game to differentiate between what the game logic needs to do and what the render renders. The three game states being used are a Start State, Play State, and Game Over State. There is also a Boolean variable checking whether a state can change at a given moment. This is necessary as the start menu is used to initialize values for the play state, and this ensures all the initializing code executes before the start state changes to play. The game state gets changed using the function shown in figure 10. This function records the previous state the game was in, in case it was needed for logic purposes.

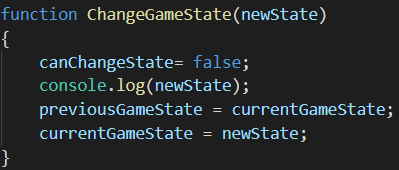


Figure - Change Game State Function

## Canvas

The canvas script only consists of two methods, the initializing method and the resize method. When the canvas is initialized it gets the canvas element from html and sets it to a js variable. Additionally, a second variable is created to hold the canvas’ context. In figure 11 the canvas context is set to a 2D context. At the end of this function, the canvas gets resized to the windows width and height.

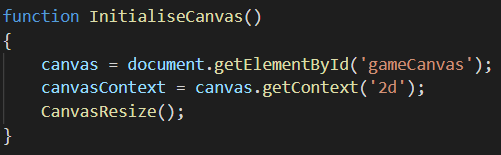


Figure - Canvas Functions

## Rendering

The rendering for the game is a state machine – shown in figure 12 - using the game state (discussed above). This function gets called in the game loop and begins by clearing anything previously rendered on the canvas, then resizing the canvas if the window size changed since the app had been opened. This behavior is required no matter what state the game is in. hence why it exists outside of the state machine. Then the state machine checks what render function to call.

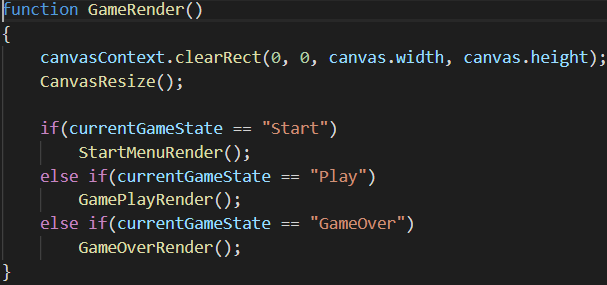


Figure - Rendering State Machine

Each state has their own function and the way these functions are structured are largely like one and other. Figure 13 shows the block of code that renders the enemy birds. If the object is stored in an array, then a for loop is used to ensure each object is renderer. Firstly, the canvas context’s current state needs to be saved and then the object specific values such as scale and rotation is applied before the sprite is rendered. If the sprite has an animation (this is discussed below) this function is called, and the canvas context is restored to its original state. The reason the context needs to be saved every time is that whenever it is restored then it delete’s the previously saved state.

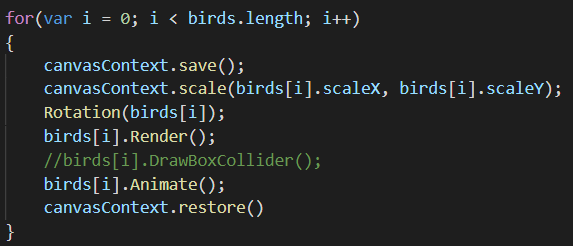


Figure - Birds Render Loop

The rotation function will rotate the birds based on their current direction of travel. Since rotation in the canvas works around the origin of the canvas as opposed to the center. To solve this the sprite would be moved to the origin before being rotated then moved back to its previous position after. However, this does not work for the rotation required for the bird. As this would flip the bird upside down and have it facing the correct direction. However, using the set transform function (shown in figure 14) the x position can be inversed and that will flip the image on the y axis. Unfortunately, this does not work well with the collider and movement, so it has been disabled for the final game.

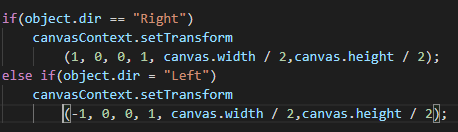


Figure - Rotation Function

## Sound

On android, an app requires the user to manually play the sound before any sound begins to play. However, this is only a problem coding sound with java script. So as a work around sound is coded in java to ensure the sound is played from the start.

The java class uses two separate built in sound classes; one for music and one for sound bites used for game feedback. The media player class is used for background music as it’s better suited for larger sound files such as music as it can be several minutes long. The code for the custom class is shown in figure 15. The apps context first gets the music file from a preset array for the instance to then set the track for the media player. The music player then sets the track to loop and plays it.

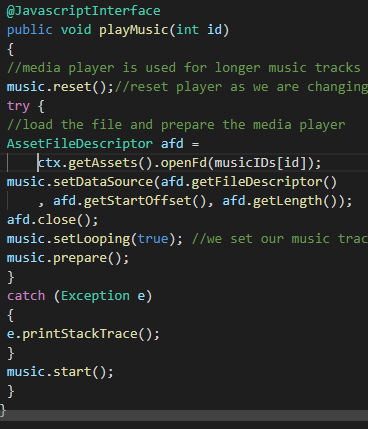


Figure - Play Music Function

The play sound function also uses the same code to load in sound files and keeps them in a separate array. A separate function also plays them using the Sound pool class. The functional difference between the Sound Pool and the Media Player classes is that the sound pool class is faster at loading files. This means it is more suitable for shorter sounds that need to be played quickly.

To use these functions in java script “@JavaScript Interface” is used.

# Inheritance

Considering several objects with different function requirements all needed to be rendered using sprites it made sense to use classes and inheritance, so all the similar requirements only needed to be coded once. This section will discuss the base class and then proceed to discuss the unique

## Sprite

The base class for all subsequent classes is the sprite class. It is because of this that the constructor for this class is the largest. The constructor is where all the values required for rendering are kept, including the image source, positions, colliders, and animation data. Getters and setters are in here for positions, and the image source. The animation function is also in here (figure 16). The first chunk of code iterates the counter variables that are used to ensure the animation transitions between frames. The second chunk of code transition the animation to the next layer of the sprite sheet. Finally, the final chunk of code checks if every frame has been rendered and loops it back to the start by resetting the counters back to 0. This animation code is based off code from an online tutorial (Malone, 2014).

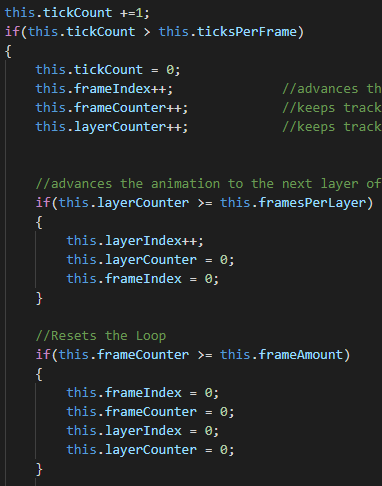


Figure - Animate Function

## Background

The background scrolls upwards from the bottom of the screen space to the top and loops around. This effect is done by using two background objects. The top of the second object touches the bottom of the first and moves up at the same speed. This is all done in the function shown in figure 17. Whenever the bottom of one of the objects is offscreen then it loops back around to the bottom of the screen.

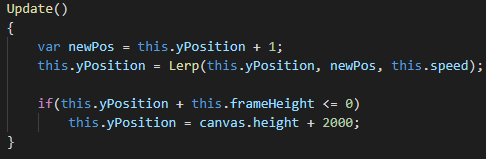


Figure - Background Update

## Player

The player class’ unique function is its move function. This takes in a string depending on what direction is passed in as a parameter. To make the movement on screen appear smoother a lerp value is calculated. This is done for every object that moves. To calculate a lerp the current position, a new position and a time variable is needed. The position moved to is the result of the lerp calculation. The 100 used to calculate the new position in figure 18 is the speed at which the player will move per frame.

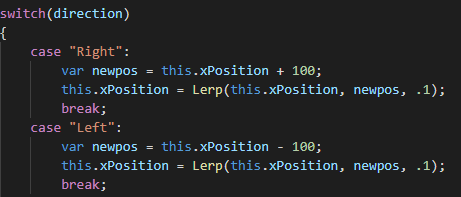


Figure - Move Function

## Enemy

The enemy has a similar move function to player with the only difference being the speed value. However, the enemy class does have two unique functions to change the rotation and direction of the birds. These are setter and getter functions. The direction sets the direction the birds travel in. this is based off the current direction. So, when the bird is travelling left, the direct gets set to right, and vice versa. Whereas the rotation gets the rotation based on the direction, as shown in figure 19. The canvas uses radians for rotation as opposed to radians. Since the rotation amount value is in degrees multiplying it using (pi/180) converts is to radians.

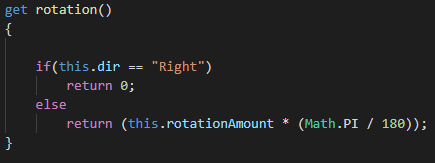


Figure – Get Direction

## Coin

The coin class uses the same above function as the enemy and player, only with different speed values again.

# Game Logic

## Initializing

The game logic script handles all the higher-level code for the rules of the game, as opposed to the lower level code in main game and other scripts that makes the game run. The initialize method for this script initializes the game objects and changes the game state to start. The initialize objects function creates new instances of classes. Classes have been used in this project to have unique objects with individual values for interaction in the game world.

## State Machine

The update function is a state machine, like the rendering update, where it checks the game state and calls the correct function for that state. The start state initializes game values if they have not already been initialized, then check for input to change the game state to play (figure 20).

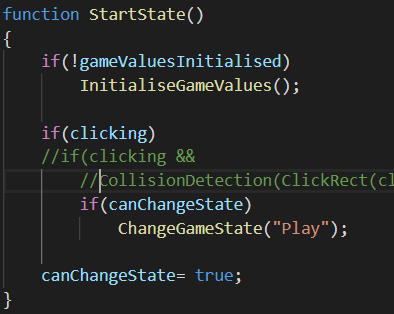


Figure - Start State Behaviour

During the Game Over state the scrolling background from the game continues to be updated, but the image source for the background is changed to the game over background. The game over state also checks for input to go back to the start state.

### Play State

Firstly, the play state sets the Boolean used to check if values initialized to false, so the start state can then initialize them or subsequent play throughs. In play state there are 6 functions that make the game run, as shown in figure 21. There are two pairs of functions that are like each other. These are the move and world check methods. Since the only difference between each pair are the game objects that are affected on one of each will be talked about.

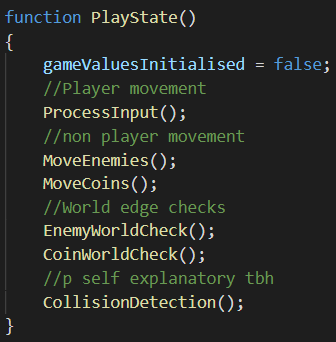


Figure - Play State Behaviour

The move functions are for loops for each non-playable object that moves each object in a direction at its individual speed. the function used to move the objects is a function in the sprite class. The birds move in two directions simultaneously (horizontally and vertically), whereas the coins only move in the vertical direction.

The world check functions check if the non-playable objects are either leaving or have left the screen space and applies behaviours accordingly. Since the position of objects is set to their top left position the objects width and height need to be taken into consideration. Figure 22 shows the check if a bird is reaching the right edge of the screen. To calculate the right side of the bird sprite the width of the frame is added to the x position. This value is then compared to the canvas’ width. However, since the bird has a scale separate to the main canvas scale the birds’ position at the edge of the screen is going to be a much larger value than the canvas width. To correct this canvas width is divided by the birds’ scale. This fixes the issue because the canvas was multiplied by the birds’ scale initially so the opposite needs to be done to get the appropriate value. The same can be done for a vertical check, although the y scale, and canvas height need to be used.

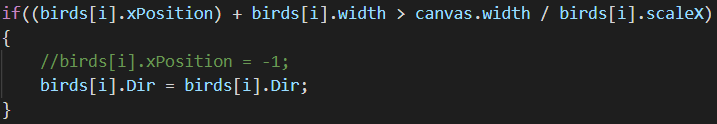


Figure - Bird Edge Check

Although the non-playable objects have functions to change their position automatically, the player object does not as the player needs to control it. This is what the process input method is used for. The input variables from main game are used here to move the player in the correct direction as shown in figure 23. This function also makes sure the player does not leave the world edge using the same logic as before.

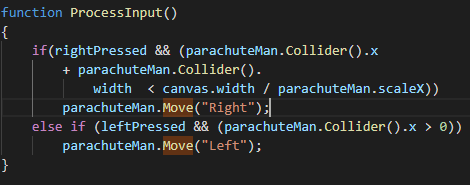


Figure - Process Input for Game Logic

Finally, collision detection is checked between the player and the non-playable objects using for loops to iterate through each array. If collision occurs, then it calls the appropriate function. The take damage function will remove an amount from the players lives and check if lives are less than 0, and if so change the state to game over. The collect coin function adds 100 onto the player score and moves the coin to a random horizontal position and puts it off screen so it looks like a new coin has spawned.

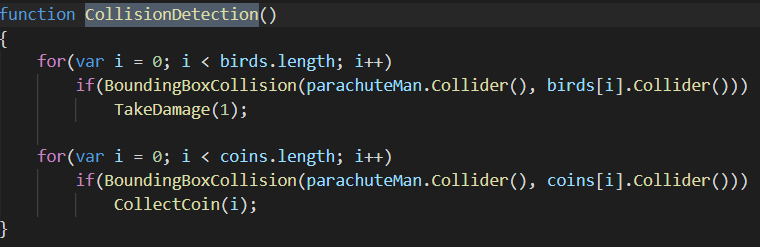
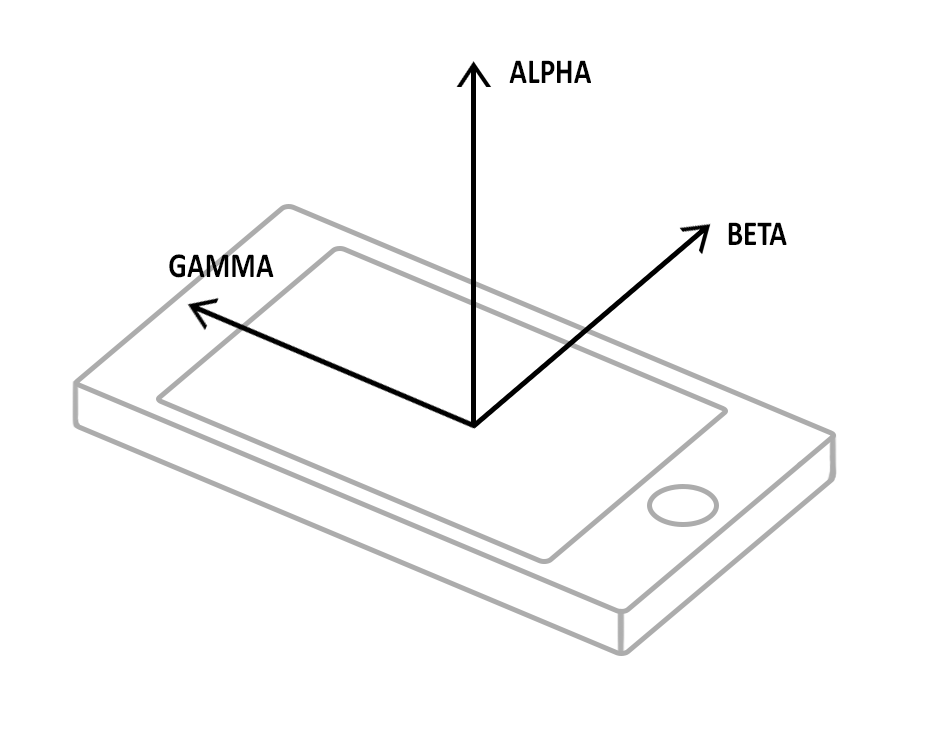


Figure - Collision Detection for Game Logic

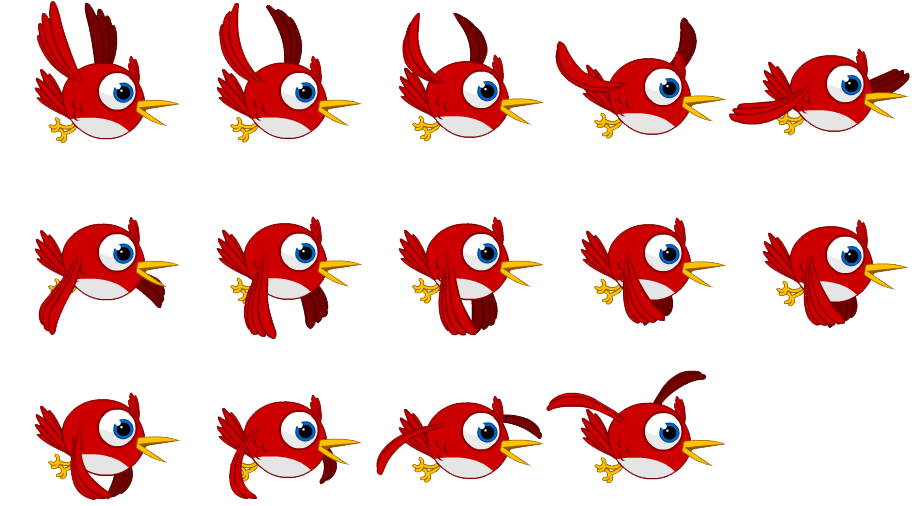
# Appendix

|  |  |  |  |
| --- | --- | --- | --- |
| **Features** | **Specification Mark** | **Extra Marks** | **Section** |
| Intro Screen | Yes | No | Game States |
| Background Music | Yes | No | Sound |
| Sound Effects | Yes | No | Sound |
| Sprites | Yes | No | Sprite |
| Background | Yes | No | Background |
| Collision Detection | Yes | No | Collision Detection |
| Touch Control | Yes | No | Controls |
| Game Over | Yes | No | Game State |
| Replay | Yes | No | Game State |
| Fonts and Text | Yes | No | Rendering |
| Motion Controls | No | Yes | Controls |
| Sprite Rotation | No | Yes | Rendering |
| Classes | No | Yes | Inheritance |
| Inheritance | No | Yes | Inheritance |
| Animation | No | Yes | Inheritance |
| Scrolling Background | No | Yes | Background (inheritance) |
| Lerp | No | Yes | Player (inheritance) |
| World Limits | No | Yes | Play State |
| World Wrap | No | Yes | Play State |

Appendix - Features List



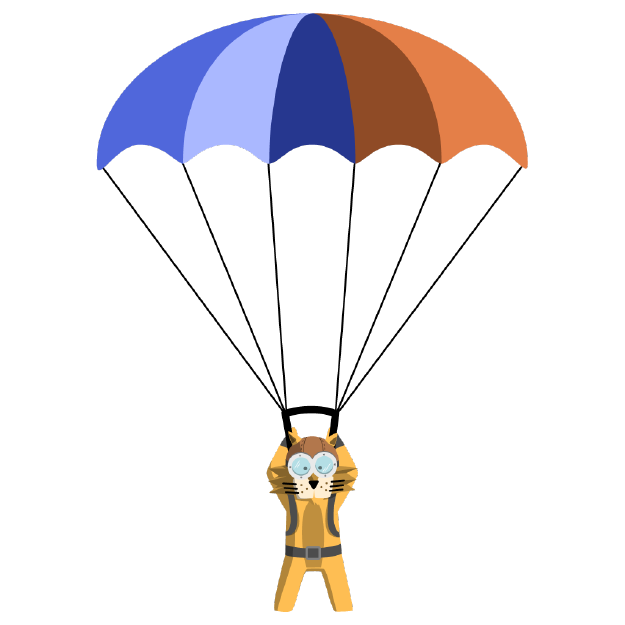
Appendix - Android rotation axes



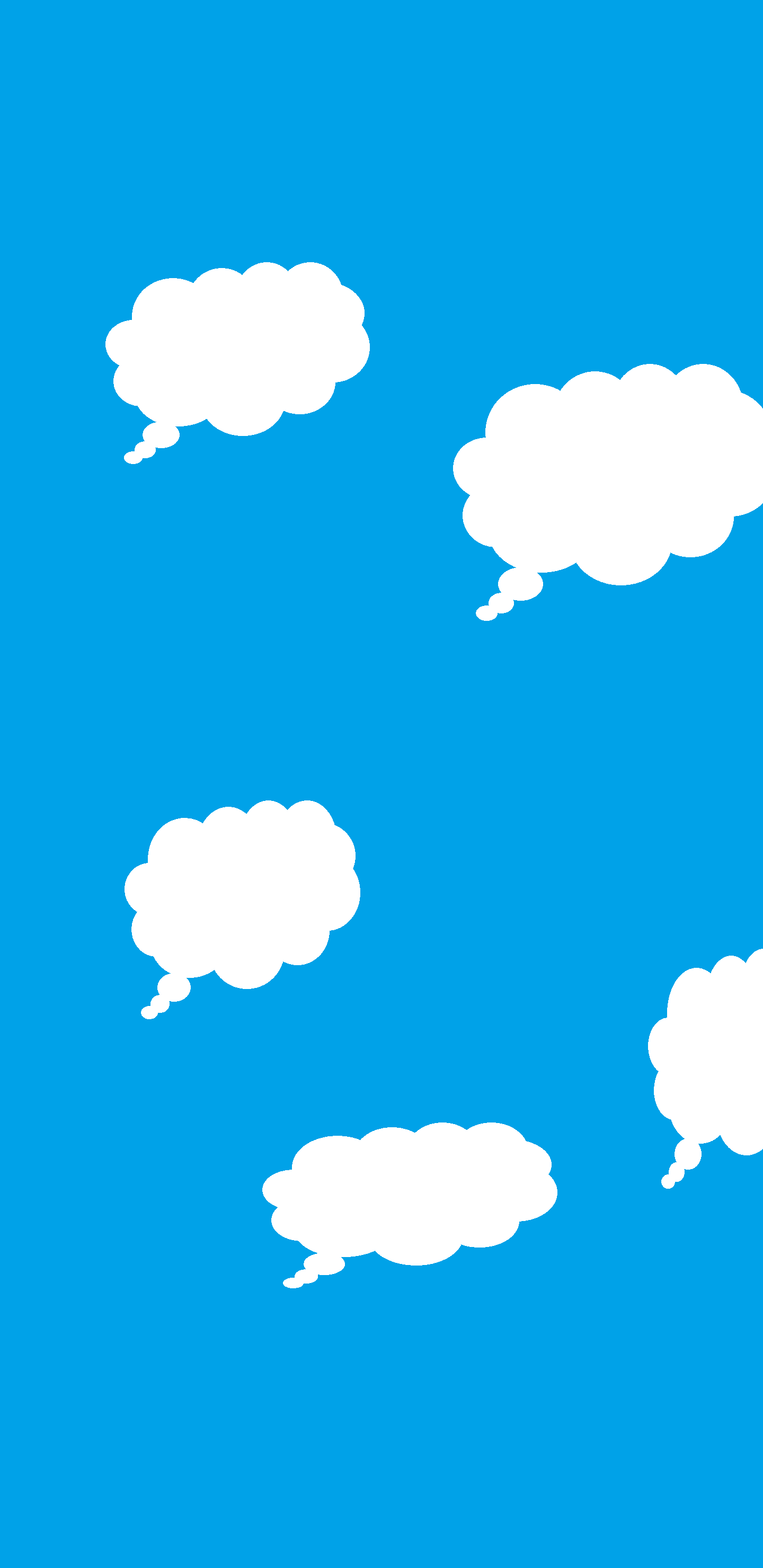
Appendix - Bird Sprite Sheet (Anonymous, n.d.)



Appendix - Coin Sprite Sheet (droptable9, 2018)



Appendix - Player Sprite (Anonymous, n.d.)



Appendix - Gameplay Background



Appendix - Game Over Screen



Appendix Start Screen

# References

Anonomys (n.d.). *Sprite Bird Animated film - sprite - Unlimited Download. Kisspng.com.*. [online] Kisspng.com. Available at: https://www.kisspng.com/png-sprite-bird-animated-film-bird-robotic-3391239/ [Accessed 11 Dec. 2018].

Deathscyp (2017). *Damage-1.wav by Deathscyp*. [online] Freesound. Available at: https://freesound.org/people/Deathscyp/sounds/404109/ [Accessed 12 Dec. 2018].

droptable9 (2018). *Spinning coin PNG from my video https://www.youtube.com/watch?v=mxMiuYm5F9Y : dogecoin*. [online] Reddit.com. Available at: https://www.reddit.com/r/dogecoin/comments/7rw8e4/spinning\_coin\_png\_from\_my\_video/ [Accessed 11 Dec. 2018].

Malone, W. (2014). Create a Sprite Animation with HTML5 Canvas and JavaScript. [online] Williammalone.com. Available at: http://www.williammalone.com/articles/create-html5-canvas-javascript-sprite-animation/?fbclid=IwAR26b0j-Z2LFm4eOwDK6Uee3h2ixgTe9RlNpoe6u90OIbxDjgnLxcNub3DI [Accessed 26 Nov. 2018].

ProjectsU012 (2016). *Coins 1 by ProjectsU012*. [online] Freesound. Available at: https://freesound.org/people/ProjectsU012/sounds/341695/ [Accessed 12 Dec. 2018].

sterferny (2017). *Country band soundcheck captured through vent.wav by sterferny*. [online] Freesound. Available at: https://freesound.org/people/sterferny/sounds/382318/ [Accessed 12 Dec. 2018].